



More

[Next Blog»](#)[phil.koopman@gmail.com](mailto:phil.koopman@gmail.com)[New Post](#)[Design](#)[Sign Out](#)

# Better Embedded System SW

Companion blog to the book **Better Embedded System Software** by Phil Koopman at Carnegie Mellon University

Showing posts with label **Essays**. [Show all posts](#)

**Monday, June 2, 2014**

## Avoid High Cyclomatic Complexity

Cyclomatic complexity is a measure of the number of paths through a particular piece of code (a higher number means the software is more complex). Critical software functions with high cyclomatic complexity are one type of "spaghetti" code that should be avoided. One can argue an exception for single-level switch statements, so long as there are no nested conditionals within the switch cases themselves.

**Consequences:** A high cyclomatic complexity for a particular function means that the function will be difficult to understand, and more difficult to test. That in turn means functions with a cyclomatic complexity above 10 or 15 can be reasonably expected to have more undetected defects than simpler functions. In practice, "spaghetti code" with a high cyclomatic complexity can reasonably be expected to have a poor design and poor unit test coverage, resulting in the reasonable expectation that it will have an elevated rate of software defects.

### **Accepted Practices:**

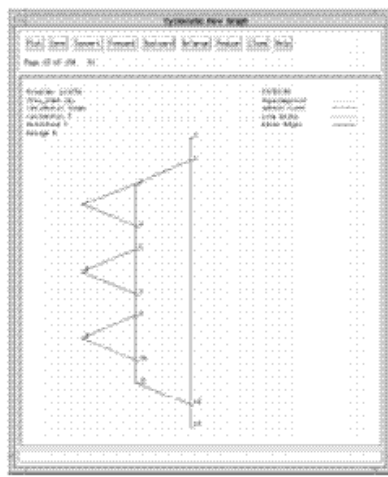
- Cyclomatic complexity should be limited for almost all software functions to a relatively small, predetermined threshold defined by the project software development plan. A range of 10 to 15 for a significant majority of functions in a project is typical.
- Functions with a cyclomatic complexity above the defined threshold of 10 to 15 should be subject to special scrutiny, and simplified if practical. Functions with a complexity significantly above 15 (perhaps a predefined threshold in the range of 30-50 and above, depending on the particular system involved) should be strictly prohibited.
- In some cases, functions with a single switch statement having many cases may be acceptable because of the regular structure of a switch statement, but should not contain nested switch statements, nor contain conditional statements within the first-level switch statement.

## **Discussion:**

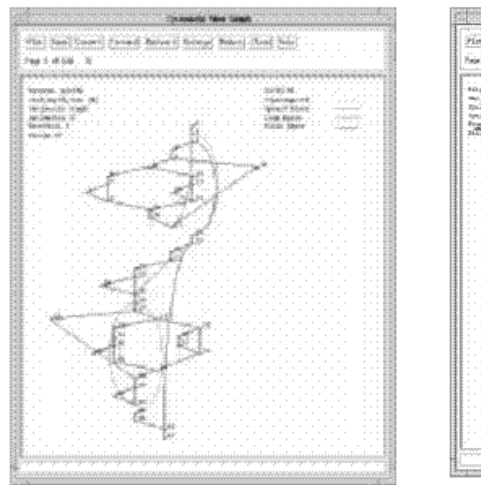
McCabe Cyclomatic Complexity (e.g., as discussed in NIST 500-235) is a measure of the complexity of a piece of software. Rather than counting the number of lines of code, cyclomatic complexity (roughly speaking) counts the number of decision points within the software. For example, a piece of software with 1000 lines of code but no “if” statements and no loops has a cyclomatic complexity of 1. But a piece of software 100 lines long might have a much higher cyclomatic complexity if it contains many levels of nested “if” statements and loops. Thus, cyclomatic complexity measures how complicated a piece of software is in terms of flow control structure rather than simply how big it is.

An important use of cyclomatic complexity is in understanding how difficult a piece of software will be to test. A common assessment of unit testing thoroughness is the “branch coverage” of a set of tests. Branch coverage measures whether tests that have been run have exercised both sides of a branch. For example, if a piece of code has a single “if ... else” construct, attaining 100% branch coverage requires one test case to exercise the “if” part, and another test case to exercise the “else” part, since a program can’t take both paths at once. But, if there are nested conditionals the number of tests required can increase dramatically due to all the different paths that might have to be taken to exercise all paths through the code. Cyclomatic complexity provides a way to understand the number of tests required for these paths. In particular, depending upon the specifics of the software, it can take up to M unit tests to achieve 100% branch coverage, where M is the McCabe Cyclomatic Complexity of the software under test. (The actual number depends upon the structure of the software, but for a higher complexity number one expects that the effort required to test the software will increase.)

Below are examples of program control flow graphs from NIST publication 500-235. The point of these figures is not the particulars of the program, but rather the visible increase in number of possible paths (from top to bottom in each graph) as the cyclomatic complexity increases. Ensuring that tests exercise each and every path through the graph is going to be a lot harder for a complexity of 17 or 28 than for a complexity of 5.



**Complexity=5**



**Complexity=17**

Cyclomatic complexity examples from NIST 500-235, 1996.

A generally accepted practice is to limit cyclomatic complexity for each software function within a system to a range of 10-15 (NIST 500-235 pg. 25). Functions having a complexity above 10 or 15 require special attention and consideration as to why the deviation is required, and can reasonably be expected to be much more difficult to test thoroughly than functions with a complexity of 10 or below. This means that the effort required to achieve high branch coverage during unit test of each function is reduced with low cyclomatic complexity.

Code with a high cyclomatic complexity normally ends up as what is referred to as “spaghetti code,” because the control flow structure is a tangled mess like a plate of spaghetti. When you pull at one end of a piece of spaghetti on a dinner plate, you have no idea which of the other ends will also move. So it is with trying to understand the details of spaghetti software. Such code has the properties of being difficult to understand, difficult to test, and more prone to defects. (Historical note: “spaghetti code” classically referred to code with many goto statements, but more recently the term has come to encompass poorly structured code in general.)

The primary exception to this rule in practice is when a “switch” statement is used to deal with a large number of alternatives in a very regular pattern, especially if the switch is being used to dispatch different command bytes from a communication protocol parser or the like. But, test effort is still significant for a switch statement with many alternative possible paths, and some compilers have trouble with switch statements having more than 256 cases, so care should be taken with large switch statements. In any event, nested conditionals should not be buried inside cases of switch statements.

### **Selected Sources:**

US National Institute of Standards and Technology (NIST)  
Special Publication 500-235 (Walsh 1996) explains the

relationship between cyclomatic complexity and testing effort. Page 25 of that publication discusses a complexity limit of 10 being an accepted practice, and says that while limits as high as 15 have been successful, even higher levels of complexity require additional testing effort.

The Reliability Information Analysis Center (“RIAC” – formerly known as “RAC”) is the focus point of the US Department of Defense for reliability information of all types. They summarize the effects of McCabe Cyclomatic Complexity (MCC) as follows: “Empirically, numbers less than ten imply reasonable structure, numbers higher than 30 are of questionable structure. Very high cyclomatic numbers of more than 50 imply the application cannot be tested, while even higher numbers of more than 75 imply that every change may trigger a ‘bad fix.’ This metric is widely used for Quality Assurance and test planning purposes.” (RAC 1996, p. 124) In context, the term “bad fix” refers in general to a situation in which every attempt to fix a software defect introduces one or more new software defects, making it impossible to remove all defects regardless of how much effort is expended.

As the number  
or program rises, the cyclomatic complexity metric rises too. Empirical  
imply reasonable structure, numbers higher than 30 are of questionable s  
matic numbers of more than 50 imply the application cannot be tested, v  
of more than 75 imply that every change may trigger a “bad fix”. This  
Quality Assurance and test planning purposes.

(RAC 1996, p

Jack Ganssle, a respected expert on embedded systems, makes it clear that “spaghetti” code must be avoided. He writes “A sure sign of disaster is the programmer who doesn’t seem to have a firm grasp of the program’s data and control flow, or who writes spaghetti code.” (Ganssle 1992, pg. 64)

Walsh found that defect rates increased significantly for modules with a McCabe Cyclomatic Complexity value above 10 on an embedded computer system (Walsh 1979, pg. 766).

Scholarly data on code complexity metrics is mixed because of the difficulty of performing controlled studies across different application areas, different programming team skill levels, different languages, and other confounding factors. For example, different projects start “counting” defects at different points in the development process, and defect counts are responsive to whether testing is thorough or not (inadequate testing may not find bugs, but that doesn’t mean the software is truly defect-free). Safety critical system developers tend to be a conservative lot (for good reason), and prefer to have less complexity because it makes it easier to inspect and test systems to ensure low defect rates.

I can also draw on my own personal experience conducting more than 130 embedded software design reviews. In essentially all cases when I see a complexity warning sign, I can find a defect within an hour (usually just minutes) of looking at code. But, more importantly, it is relatively straightforward to make the code less complex and more understandable (and therefore less defect prone) with very little effort in almost every case of high complexity that I have encountered. And, usually, once I show the programming team how to simplify, they agree it is better and start the practice themselves on other code. So, in my own extensive consulting and teaching experience, what I have found is that code with complexity warning signs is indicative of a software team that needs to improve in some significant way to create acceptable software – which is an undesirable state for developers of safety critical systems. When I find code that is simple, understandable, and devoid of unnecessary complexity, I know that I have found programmers who in general know what they are doing.

The developers of safety critical software standards feel the same way. They make it clear that complexity must be kept low in safety critical systems. Simplicity is one of the tools that is essential for creating safe systems. All of MISRA Report 5 is devoted to the topic of Software Metrics. It describes MCC (MISRA Report 5, pp. 37-38, p. 60) and gives a maximum acceptable complexity of 15 for safety critical systems.

The FAA recommends using a complexity metric such as MCC (FAA 2000, pg. J-17, J-23). NASA recommends reducing complexity using MCC and design for modularity via having weak coupling (NASA 2004, p. 95).

There are many possible complexity metrics for code. I have found that cyclomatic complexity is a useful predictor of likely code quality.

### **References:**

- FAA, System Safety Handbook, Appendix J: Software Safety, Federal Aviation Administration, Dec. 2000
- Ganssle, J., The Art of Programming Embedded Systems, Academic Press, 1992.
- MISRA, Report 5: Software Metrics, February 1995.
- NASA-GB-8719.13, NASA Software Safety Guidebook, NASA Technical Standard, March 31, 2004.
- RAC, Introduction to Software Reliability: a state of the art review, Reliability Analysis Center, 1996.
- Watson, A. & McCabe, T., (NIST 500-235), Structured Testing: a testing methodology using the cyclomatic complexity metric, Special Publication 500-235, National Institute of Standards and Technology, Sept. 1996

Posted by Phil Koopman at [Monday, June 02, 2014](#) 2 comments  

Labels: [coding style](#), [Essays](#), [metrics](#)

**Monday, May 26, 2014**

## **Proper Watchdog Timer Use**

Watchdog timers are a prevalent mechanism for helping to ensure embedded system reliability. But they only work if you use them properly. Effective watchdog timer use requires that the failure of any periodic task in the system must result in a watchdog timer reset.

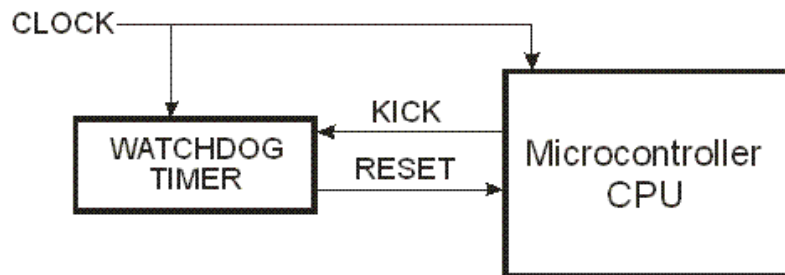
**Consequences:** Improper use of a watchdog timer leads to a false sense of security in which software task deaths and software task time overruns are not detected, causing possible missed deadlines or partial software failures.

### **Accepted Practices:**

- If multiple periodic tasks are in the system, each and every such task must contribute directly to the watchdog being kicked to ensure every task is alive.
- Use of a hardware timer interrupt to directly kick the watchdog is a bad practice. (There is arguably an exception of the ISR keeps a record of all currently live tasks as described later.)
- Inferring task health by monitoring the lowest priority task alone is a bad practice. This approach fails to detect dead high priority tasks.
- The watchdog timeout period should be set to the shortest practical value. The system should remain safe even if any combination of tasks dies for the entire period of the watchdog timeout value.
- Every time the watchdog timer reset happens during testing of a fully operational system, that fact should be recorded and investigated.

### **Discussion**

Briefly, a watchdog timer can be thought of as a counter that starts at some predetermined value and counts down to zero. If the watchdog actually gets to zero, it resets the system in the hopes that a system reset will fix whatever problem has occurred. Preventing such a reset requires “kicking” (terms for this vary) the watchdog periodically to set the count back at the original value, preventing a system reset. The idea is for software to convince the hardware watchdog that the system is still alive, forestalling a reset. The idea is not unlike asking a teenager to call in every couple hours on a date to make sure that everything is going OK.



Watchdog timer arrangement.

Once the watchdog kicks a system reset is the most common reaction, although in some cases a permanent shutdown of the system is preferable if it is deemed better to wait for maintenance intervention before attempting a restart.

Getting the expected benefit from a watchdog timer requires using it in a proper manner. For example, having a hardware timer interrupt trigger unconditional kicking of the watchdog is a specifically bad practice, because it doesn't indicate that any software task except the hardware timer ISR is working properly. (By analogy, having your teenager set up a computer to automatically call home with a prerecorded "I'm OK" message every hour on Saturday night doesn't tell you that she's really OK on her date.)

For a system with multiple tasks it is essential that each and every task contribute to the watchdog being kicked. Hearing from just one task isn't enough – all tasks need to have some sort of unanimous "vote" on the watchdog being kicked.

Correspondingly, a specific bad practice is to have one task or ISR report in that it is running via kicking the watchdog, and infer that this means all other tasks are executing properly. (Again by analogy, hearing from one of three teenagers out on different dates doesn't tell you how the other two are doing.) As an example, the watchdog "should never be kicked from an interrupt routine" (MISRA Report 3, p. 38), which in general refers to the bad practice of using a timer service ISR to kick the watchdog.

A related bad practice is assuming that if a low priority task is running, this means that all other tasks are running. Higher priority tasks could be "dead" for some reason and actually give more time for low priority tasks to run. Thus, if a low priority task kicks the watchdog or sets a flag that is the sole enabling data for an ISR to kick the watchdog, this method will fail to detect if other tasks have failed to run in a timely periodic manner.

Monitoring CPU load is not a substitute for a watchdog timer. [Tasks can miss their deadlines even with CPU loads of 70%-80%](#) because of bursts of momentary overloads that are to be expected in a real time operating system environment as a normal part of system operation. For this reason, another bad practice is using software inside the system being monitored to

perform a CPU load analysis or other indirect health check and kick the watchdog periodically by default unless the calculation indicates a problem. (This is a variant of kicking the watchdog from inside an ISR.)

The system software should not be in charge of measuring workload over time -- that is the job of the watchdog. The software being monitored should kick the watchdog if it is making progress. It is up to the watchdog mechanism to decide if progress is fast enough. Thus, any conditional watchdog kick should be done just based on liveness (have tasks actually been run), and not on system loading (do we think tasks are probably running).

One way to to kick a watchdog timer in a multi-tasking system is sketched by the below graphic:

## Effective Multi-Tasking Watchdog Ap

```
void task0(void) { .. Do stuff..; Alive(0x1);
void task1(void) { .. Do stuff..; Alive(0x2);
void task2(void) { .. Do stuff..; Alive(0x4);
void task3(void) { .. Do stuff..; Alive(0x8);
```

### ◆ Main idea – each task sets a bit indicating it has run

- Separate watchdog monitor task kicks watchdog only when every
- Needs to be modified to account for task periods, but this is the b

```
static uint16 watch_flag = 0;
void Alive(uint16 x)
{ DisableInterrupts();
  watch_flag |= x;
  EnableInterrupts();
} // set task's "I'm Alive" bit

void taskw(void) // run periodically
{ if (watch_flag == 0x0F) // if all tasks alive
  { kick(); // kick watchdog
    watch_flag = 0; // erase flags
  }
}
```

Key attributes of this watchdog approach are: (1) all tasks must be alive to kick the WDT. If even one task is dead the WDT will time out, resetting the system. (2) The tasks do not keep track of time or CPU load on their own, making it impossible for them to have a software defect or execution defect that “lies” to the WDT itself about whether things are alive. Rather than making the CPU’s software police itself and shut down to await a watchdog kick if something is wrong, this software merely has the tasks report in when they finish execution and lets the WDT properly due its job of policing timeliness. More sophisticated versions of this code are possible depending upon the system involved; this

is a classroom example of good watchdog timer use. Where "taskw" is run from depends on the scheduling strategy and how tight the watchdog timer interval is, but it is common to run it in a low-priority task.

Setting the timing of the watchdog system is also important. If the goal is to ensure that a task is being executed at least every 5 msec, then setting the watchdog timer at 800 msec doesn't tell you there is a problem until that task is 795 msec late. Watchdog timers should be set reasonably close to the period of the slowest task that is kicking them, with just a little extra time beyond what is required to account for execution variation and scheduling jitter.

If watchdog timer resets are seen during testing they should be investigated. If an acceptable real time scheduling approach is used, a watchdog timer reset should never occur unless there has been system failure. Thus, finding out the root cause for each watchdog timer reset recorded is an essential part of safety critical design. For example, in an automotive context, any watchdog timer event recordings could be stored in the vehicle until it is taken in for maintenance. During maintenance, a technician's computer should collect the event recordings and send them back to the car's manufacturer via the Internet.

While watchdog timers can't detect all problems, a good watchdog timer implementation is a key foundation of creating a safe embedded control system. It is a negligent design omission to fail to include an acceptable watchdog timer in a safety critical system.

### **Selected Sources**

Watchdog timers are a classical approach to ensuring system reliability, and are a pervasive hardware feature on single-chip microcontrollers for this reason.

An early scholarly reference is a survey paper of existing approaches to industrial process control (Smith 1970, p. 220). Much more recently, Ball discusses the use of watchdog timers, and in particular the need for every task to participate in kicking the watchdog. (Ball 2002, pp 81-83). Storey points out that while they are easy to implement, watchdog timers do have distinct limitations that must be taken into account (Storey pg. 130). In other words, watchdog timers are an important accepted practice that must be designed well to be effective, but even then they only mitigate some types of faults.

Lantrip sets forth an example of how to ensure multiple tasks work together to use a watchdog timer properly. (Lantrip 1997). Ganssle discusses how to arrange for all tasks to participate in kicking the watchdog, ensuring that some tasks don't die while others stay alive. (Ganssle 2000, p. 125).

Brown specifically discusses good and bad practices. “I’ve seen some multitasking systems that use an interrupt to tickle the watchdog. This approach defeats the whole purpose for having one in the first place. If all the tasks were blocked and unable to run, the interrupt method would continue to service the watchdog and the reset would never occur. A better solution is to use a separate monitor task that not only tickles the watchdog, but monitors the other system tasks as well.” (Brown 1998 pg. 46).

The MISRA Software Guidelines recommend using a watchdog to detect failed tasks (MISRA Report 1, p. 43), noting that tasks (which they call “processes”) may fail because of noise/EMI, communications failure, software defects, or hardware faults. The MISRA Software Guidelines say that a “watchdog is essential, and must not be inhibited,” while pointing out that having returning an engine to idle in a throttle-by-wire application could be unsafe. (MISRA Report 1, p. 49). MISRA also notes that “The consequence of each routine failing must be identified, and appropriate watchdog and default action specified.” (MISRA Report 4 p. 33, emphasis added)

NASA recommends using a watchdog and emphasizes that it must be able to detect death of all tasks (NASA 2004, p. 93). IEC 61508-2 lists a watchdog timer as a form of test by redundant hardware (pg. 115) (without implying that it provides complete redundancy).

Addy identified a task death failure mode in a case study (Addy 1991, pg. 79) due to a task encountering a run-time fault that was not properly caught, resulting in the task never being restarted. Thus, it is reasonably conceivable that a task will die in a multitasking operating system. Inability to detect a task death is a defect in a watchdog timer, and a defective watchdog timer approach undermines the safety of the entire system. With such a defective approach, it would be expected that task deaths or other foreseeable events will go undetected by the watchdog timer.

## **References:**

- Addy, E., A case study on isolation of safety-critical software, Proc. Conf Computer Assurance, pp. 75-83, 1991.
- Ball, Embedded Microprocessor Systems: Real World Design, Newnes, 2002.
- Brown, D., “Solving the software safety paradox,” Embedded System Programming, December 1998, pp. 44-52.
- Ganssle, J., The Art of Designing Embedded Systems, Newnes, 2000.
- IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES), International Electrotechnical Commission, 1998.

- Lantrip, D. & Bruner, L., General purpose watchdog timer component for a multitasking system, Embedded Systems Programming, April 1997, pp. 42-54.
- MISRA, Report 1: Diagnostics and Integrated Vehicle Systems, February 1995.
- MISRA, Report 3: Noise, EMC and Real-Time, February 1995.
- MISRA, Report 4: Software in Control Systems, February 1995.
- NASA-GB-8719.13, NASA Software Safety Guidebook, NASA Technical Standard, March 31, 2004.
- Smith, Digital control of industrial processes, ACM Computing Surveys, Sept. 1970, pp. 211-241.
- Storey, N., Safety Critical Computer Systems, Addison-Wesley, 1996.

Posted by Phil Koopman at [Monday, May 26, 2014](#) 15 comments    
Labels: [Essays](#), [safety](#), [watchdog timer](#)

**Monday, May 12, 2014**

## **Real Time Scheduling Analysis for Critical Systems**

Critical embedded software must be created using methodical scheduling analysis that ensures that every task can meet its deadline under worst-case fault-free operational conditions. Looking just at CPU load is not only inadequate to determine if you will meet real time deadlines -- it is a specifically bad practice that is to be avoided.

### **Consequences:**

Failing to do an accurate real time schedule design for a safety critical system means that designers do not know if the system will miss deadlines. It is impractical in a complex system (which means almost any real-world product) to do enough testing with a multitasked RTOS-based system to ensure that deadlines will be met under worst case conditions. Thus, a design team that fails to do scheduling analysis should appreciate that they have an unknown probability of missing deadlines, and can reasonably expect to miss deadlines in the worst case if the CPU load is above about 69% for a large number of non-harmonic tasks.

### **Accepted Practices:**

- Using Rate Monotonic Scheduling (RMA) when using an RTOS with multi-tasking capability, or using some other mathematically sound scheduling analysis. Rate Monotonic Scheduling (RMS) is what you get as a result of RMA.

- Documenting the RMA analysis including at least listing: all tasks, WCET for each task, period of each task, and worst case system blocking time. Assumptions used in the analysis should be stated and justified (for example, an assumption that no low priority task blocks a high priority task must be justified via written explanation, or if untrue then advanced techniques must be used to ensure schedulability). The system use must be less than 100% of the CPU if a set of favorable assumptions such as harmonic task periods can be documented to hold, and may be restricted to as low as 69.3% CPU used (meaning 30.7% unused CPU capacity) in the general case.
- A specifically *bad* practice is basing real time performance decisions solely on spare capacity (e.g., “CPU is only 80% loaded on average”) in the absence of mathematical scheduling analysis, because it does not guarantee safety critical tasks will meet their deadlines. Similarly, monitoring spare CPU capacity as the only way to infer whether deadlines are being met is a specifically *bad* practice, because it does not actually tell you whether high frequency deadlines are being met or not.
- Permitting more than one instance of a real-time task to queue is a specifically *bad* practice, because this can only happen when real time deadlines are being missed. This practice is a bandaid for a real-time system, and indicates that the system is missing its real-time deadlines.

## Discussion

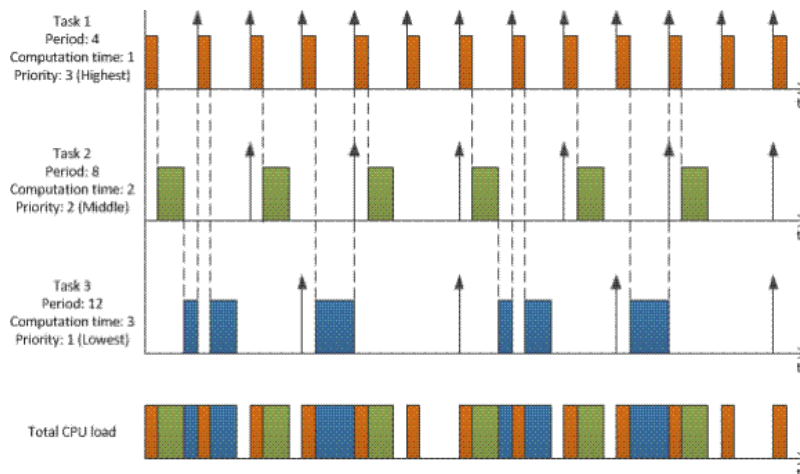
Real time scheduling is a mathematical approach to ensuring that every task in a real time embedded system meets its deadlines under all specified operating conditions. Using a mathematical approach is required because testing can only exercise some of the system operating conditions. There are almost always real time scheduling situations that can't be adequately tested in a complex piece of software, requiring either simplifying the system to make testing feasible, or using a rigorous mathematical approach to ensure that complex scheduling is guaranteed to work. When a multi-tasking real time operating system such as OSEK is used, a mathematical approach must be used to ensure deadlines are met.

## Rate Monotonic Analysis

The generally accepted method for scheduling critical real time systems is to perform Rate Monotonic Analysis (RMA), possibly with one of a number of adaptations for special circumstances, and create a system with a Rate Monotonic Schedule. RMA has the virtue of providing a simple set of rules that guarantees all tasks in a system will meet their deadlines. To achieve this, RMA requires rules to be followed such as all tasks having a defined fastest time period at which they will run, and the period of tasks being harmonic multiples to permit using 100% of the CPU capacity. (Task periods are considered harmonic if they are exact

multiples of each other. For example, periods of 1, 10, 100, 1000 msec are harmonic, but 1, 9, 98, 977 msec are not harmonic.) To implement RMA, designers sort tasks based on period, and assign the fastest task to the highest priority, second fastest task to the second highest priority, and so on. If a designer wishes to have multiple tasks at the same priority that is acceptable, but it is required that all tasks at a given priority execute at the same period.

The benefit of RMA is that there is a mathematical proof that all tasks will meet deadlines if a certain set of rules is followed. If any of the rules is bent, however, then more complex mathematical analysis must be performed or other special techniques used to ensure that deadlines will be met. In particular, if task periods are not harmonic multiples of each other, guaranteeing that deadlines will be met requires leaving slack capacity on the CPU. For a system with many tasks, RMA can only guarantee that task deadlines will be met if the CPU load is a maximum of 69.3%. (For this reason, it is common for embedded systems to use harmonic multiples of task periods.)



Example execution of a system scheduled with RMA.

(Source: <http://blogs.abo.fi/alexeevpetr/2011/11/24/generating-multithread-code-from-simulink-model-for-embedded-target/>)

Employing RMA requires at least the following steps at design time: listing all tasks in the system including interrupt service routines, determining the fastest period of each task, determining the worst case execution time (WCET) of each task, and determining the maximum blocking time of the system (longest time during which interrupts are disabled). Given those numbers, tasks can be assigned priorities and designers can know if the system will always meet its deadlines. Given correct RMA scheduling and a fault-free operational system, every task will complete by the end of its period, so it will never be possible to have a second occurrence of a task enqueued while waiting for a first occurrence to complete.

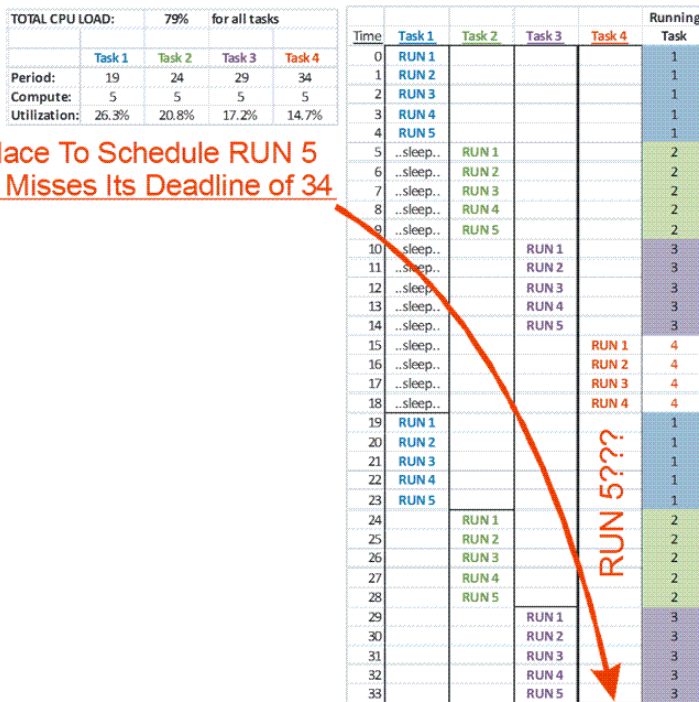
If the CPU is over-subscribed, there are established methods of

ensuring that critical tasks always complete while non-critical tasks get whatever CPU resources are left. The simplest method is to simply assign all non-critical tasks priorities lower than the lowest priority critical task (which works so long as those non-critical tasks can't substantively delay with the operation of the critical tasks). If it is important to share available CPU time across a number of noncritical tasks even when the CPU is overloaded, this can be done by having non-critical tasks take turns at the lowest priority. Note that an overloaded CPU does not cause any critical tasks to miss their deadlines in this scenario; it is simply a matter of making non-critical tasks wait a bit longer to execute if the CPU is overloaded. (This assumes that the CPU can handle worst case demand from critical tasks. This assumption must be ensured by the design process for the system to be safe.)

### **Less Than 100% CPU Load Does Not Guarantee Deadlines Are Met**

*A specifically bad practice* is looking at idle time during testing to determine whether or not the CPU is overloaded, and inferring from less than 100% usage that the system will meet its deadlines. That simply does not account for the worst case, or potentially even just an infrequent heavy load case.

As a non-computer example of missing deadlines with less than 100% loading, let's say you want to spend five days per week working and three days out of every 12 volunteering at a homeless shelter (the fact that it is out of 12 days instead of out of 14 days makes these periods non-harmonic). Because work has a period of 7 days, it will have higher priority than the 12 day volunteer service period. This means you'll complete all of your work the first 5 days (Monday – Friday) out of each 7-day work week before you start volunteering on weekends. Most weeks this will work fine (you'll have enough time for both). But when the 12-day volunteer period starts on a Monday, you'll only have one weekend (two days) in the 12 day period for volunteering that runs Monday of one week through Friday of the next week. Thus you'll be a day short for volunteering whenever the volunteer period starts on a Monday. The amount of time you've committed is 5 days out of 7 plus 3 days out of 12:  $5/7 + 3/12 = 96.4\%$ . But you're going to come up a whole day short on volunteering once in a while even though you are less than 100% committed and you are taking some weekend days off, performing neither task. You could solve this by committing 4 days out of 14 to volunteering, which is actually a slightly higher workload (100% total), but changes the period to be harmonic with the weekly work cycle. (3.5 days out of every 14 would also work.) Thus, as shown by this example, non-harmonic task periods can result in missed deadlines even with a workload that is less than 100%.



Example of 79% loaded CPU missing a task deadline with only 4 tasks.

As a computer-based example, the above figure shows another example in terms of a task in a four-task system missing its deadline on a system with non-harmonic periods even though the CPU is only 79% loaded. This scenario would happen rarely, and only when all the tasks' periods synchronize, which for this example would be only once every Least Common Multiple of  $(19, 24, 29, 34) = 224,808$  time units – and even then only if each task actually took its worst case time to execute. Thus, while Task 4 would be expected to miss its deadline in operation, detecting that situation might be difficult with limited duration testing.

## Selected Sources

Rate Monotonic Scheduling was developed to address the problem of creating an easy-to-follow recipe for ensuring that real time schedules can be met in a system with multiple tasks. Influential early papers include (Liu & Layland 1973 and Lehoczky et al. 1989).

Ganssle recommends the use of RMA as early as 1992 (Ganssle 1992, pg. 200-201) and sketches its use, giving a reference for more details.

Douglass provides a pattern for static priority real time scheduling and states: “the most common policy for the selection of policies is rate monotonic scheduling or RMS”. (Douglass 2002, section 5.9.5; note that rate monotonic scheduling is what you get as a result of rate monotonic analysis).

In the context of safety critical operating systems, Kleidermacher says “Rate monotonic analysis (RMA) is frequently used by

system designers to analyze and predict the timing behavior of systems.” (Kleidermacher 2001, pg. 30).

MISRA Report 3 discusses the use of real-time kernels (which for our purposes are operating systems that use some sort of scheduling approach). It notes that there are a number accepted scheduling techniques, and that the use of “best available technology” such as for example using a certified RTOS brings benefit, providing a reference to a number of text.

### **References:**

- Douglass, Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns, Addison-Wesley Professional, 1999.
- Ganssle, J., The Art of Programming Embedded Systems, Academic Press, 1992.
- Kleidermacher, D. & Griglock, M., Safety-Critical Operating Systems, Embedded Systems Programming, Sept. 2001, pp. 22-36.
- Lehoczky, J.; Sha, L.; Ding, Y. "The rate monotonic scheduling algorithm: exact characterization and average case behavior", IEEE Real-Time Systems Symposium, 1989, pp. 166–171.
- Liu, C. L.; Layland, J., "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, 1973, (1): 46–61
- MISRA, Report 3: Noise, EMC and Real-Time, February 1995.

Posted by Phil Koopman at [Monday, May 12, 2014](#) 4 comments  

Labels: [Essays](#), [real time](#)

**Monday, May 5, 2014**

## **Mitigating Data Corruption**

As previously discussed, data corruption from bit flips can happen from a variety of sources. The results from such faults can be catastrophic. An effective technique should be used to detect both software- and hardware-caused corruption of critical data. Effective techniques include error coding and data replication in various forms.

### **Consequences:**

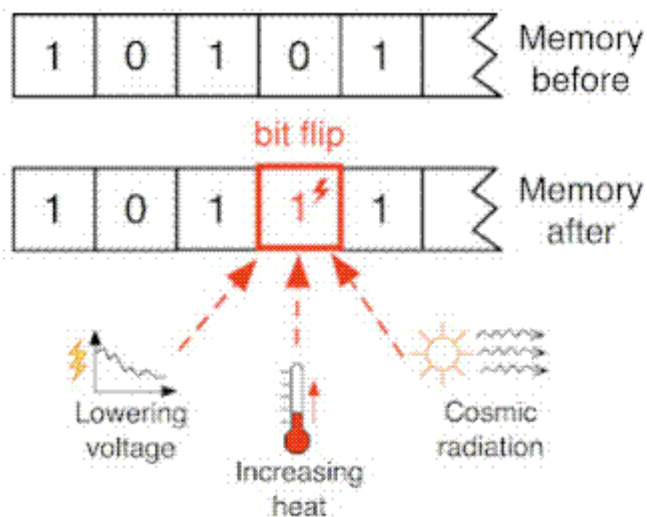
Unintentional modification of data values can cause arbitrarily bad system behavior, even if only one bit is modified. Therefore, safety critical systems should take measures to prevent or detect such corruptions, consistent with the estimated risk of hardware-induced data corruption and anticipated software quality. Note that even best practices may not prevent corruption due to software concurrency defects.

### **Accepted Practices:**

- When hardware data corruption may occur and automatic error correction is desired, use a hardware single error correction/multiple error detection circuitry (SECMED) form of Error Detection and Correction circuitry (EDAC), sometimes just called Error Correcting Code circuitry (ECC) for all bytes of RAM. This would protect against hardware memory corruption, including hardware corruption of operating system variables. However, it does not protect against software memory corruption.
- Use a software approach such as a [cyclic redundancy code CRC \(preferred\)](#), or checksum to detect a corrupted program image, and test for corruption at least when the system is booted.
- Use a software approach such as keeping redundant copies to detect software data corruption of RAM values.
- Use fault injection to test data corruption detection and correction mechanisms.
- Perform memory tests to ensure there are no hard faults.

### **Discussion:**

Safety critical systems must protect against data corruption to avoid small changes in data which can render the system unsafe. Even a single bit in memory changing in the wrong way could cause a system to change from being safe to unsafe. To guard against this, various schemes for memory corruption detection and prevention are used.



Effect of a bit flip (Source:

<http://dependablesystem.blogspot.com/2012/05/flip-happens.html>)

### **Hardware and Software Are Both Corruption Sources**

Hardware memory corruption occurs when a radiation event, voltage fluctuation, source of electrical noise, or other cause makes one or more bits flip from one value to another. In non-volatile memory such as flash memory, wearout, low programming voltage, or electrical charge leakage over time can

also cause bits in memory to have an incorrect value. Mitigation techniques for these types of memory errors include the use of hardware error detection/correction codes (sometimes called “EDAC”) for RAM, and typically the use of a “checksum” for flash memory to ensure that all the bytes, when “added up,” give the same total as they did when the program image was stored in flash.

If hardware memory error detection support is not available, RAM can also be protected with some sort of redundant storage. A common practice is to store two copies of a value in two different places in RAM, often with one copy inverted or otherwise manipulated. It's important to avoid storing the two copies next to each other to avoid problems of errors that corrupt adjacent bits in memory. Rather, there should be two entirely different sections of memory for mirrored variables, with each section having only one copy of each mirrored variable. That way, if a small chunk of memory is arbitrarily corrupted, it can at most affect one of the two copies of any mirrored variable. Error detection codes such as checksums can also be used, and provide a tradeoff of increased computation time when a change is made vs. requiring less storage space for error detection information as compared to simple replication of data.

Software memory corruption occurs when one part of a program mistakenly writes data to a place that should only be written to by another part of the program due to a software defect. This can happen as a result of a defect that produces an incorrect pointer into memory, due to a buffer overflow (e.g., trying to put 17 bytes of data into a 16 byte storage area), due to a stack overflow, or due to a concurrency defect, among other scenarios.

Hardware error detection does not help in detecting software memory corruption, because the hardware will ordinarily assume that software has permission to make any change it likes. (There may be exceptions if hardware has a way to “lock” portions of memory from modifications, which is not the case here.)

Software error detection may help if the corruption is random, but may not help if the corruption is a result of defective software following authorized RAM modification procedures that just happen to point to the wrong place when modifications are made. While various approaches to reduce the chance of accidental data corruption can be envisioned, acceptable practice for safety critical systems in the absence of redundant computing hardware calls for, at a minimum, storing redundant copies of data. There must also be a recovery plan such as system reboot or restoration to defaults if a corruption is detected.

### **Data Mirroring**

A common approach to providing data corruption protection is to use a data mirroring approach in which a second copy of a variable having a one's complement value is stored in addition to the ordinary variable value. A one's complement representation

of a number is computed by inverting all the bits in a number. So this means one copy of the number is stored normally, and the second copy of that same number is stored with all the bits inverted (“complemented”). As an example, if the original number is binary “0000” the one’s complement mirror copy would be “1111.” When the number is read, both the “0000” and the “1111” are read and compared to make sure they are exact complements of each other. Among other things, this approach gives protection against a software defect or hardware corruption that sets a number of RAM locations to all be the same value. That sort of corruption can be detected regardless of the constant corruption value put into RAM, because two mirrored copies can’t have the same value unless at least one of the pair has been corrupted (e.g., if all zeros are written to RAM, both copies in a mirrored pair will have the value “0000,” indicating a data corruption has occurred).

Mirroring can also help detect hardware bit flip corruptions. A bit flip is when a binary value (a 0 or 1), is corrupted to have the opposite value (changing to a 1 or 0 respectively), which in turn corrupts the value of the number stored at the memory location suffering one or more bit flips. So long as only one of two mirror values suffers a bit flip, that corruption will be detectable because the two copies won’t match properly as exact complements of each other.

A good practice is to ensure that the mirrored values are not adjacent to each other so that an erroneous multi-byte variable update is less likely to modify both the original and mirrored copy. Such mirrored copies are vulnerable to a pair of independent bit flips that just happen to correspond to the same position within each of a pair of complemented stored values. Therefore, for highly critical systems a Cyclic Redundancy Check (CRC) or other more advanced error detection method is recommended.

It is important to realize that all memory values that can conceivably cause a system hazard need to be protected by mirroring, not just a portion of memory. For example a safety-critical Real Time Operating System will have values in memory that control task scheduling. Corruption of these variables can lead to task death or other problems if the RTOS doesn't protect data integrity, even if the application software does use mirroring. Note that there are multiple ways for an RTOS to protect its data integrity from software and hardware defects beyond this, such as via using hardware access protection. But, if the only mechanism being used in a system to prevent memory corruption is mirroring, the RTOS has to use it too or you have a vulnerability.

## **Selected Sources**

Automotive electronics designers knew as early as 1982 that data corruption could be expected in automotive electronics. Seeger

writes: “Due to the electrically hostile environment that awaits a microprocessor based system in an automobile, it is necessary to use extra care in the design of software for those systems to ensure that the system is fault tolerant. Common faults that can occur include program counter faults, *altered RAM locations*, or erratic sensor inputs.” (Seeger 1982, abstract, emphasis added). Automotive designers generally accepted the fact that RAM location disruptions would happen in automotive electronics (due to electromagnetic interference (EMI), radiation events, or other disturbances), and had to ensure that any such disruption would not result in an unsafe system.

Stepner, in a paper on real time operating systems that features a discussion of OSEK (the trade name of an automotive-specific real time operating system), states with regard to avoiding corruption of data: “One technique is the redundant storage of critical variables, and comparison prior to being used. Another is the grouping of critical variables together and keeping a CRC over each group.” (Stepner 1999, pg. 155).

Brown says “We’ve all heard stories of bit flips that were caused by cosmic rays or EMI” and goes on to describe a two-out-of-three voting scheme to recover from variable corruption. (Brown 1998 pp. 48-49). A variation of keeping only two copies permits detection but not correction of corruption. Brown also acknowledges that designers must account for software data corruption, saying “Another, and perhaps more common, cause of memory corruption is a rogue pointer, which can run wild through memory leaving a trail of corrupted variables in its wake. Regardless of the cause, the designer of safety-critical software must consider the threat that sometime, somewhere, a variable will be corrupted.” (id., p. 48).

Kleidermacher says: “When all of an application’s threads share the same memory space, any thread could—intentionally or unintentionally— corrupt the code, data, or stack of another thread. A misbehaved thread could even corrupt the kernel’s own code or internal data structures. It’s easy to see how a single errant pointer in one thread could easily bring down the entire system, or at least cause it to behave unexpectedly.” (Kleidermacher 2001, pg. 23). Kleidermacher advocates hardware memory protection, but in the absence of a hardware mechanism, software mechanisms are required to mitigate memory corruption faults.

Fault injection is a way to test systems to see how they respond to faults in memory or elsewhere (see also an upcoming post on that topic). Fault injection can be performed in hardware (e.g., by exposing a hardware circuit to a radiation source or by using hardware test features to modify bit values), or injected via software means (e.g., slightly modifying software to permit flipping bits in memory to simulate a hardware fault). In a research paper, Vinter used a hybrid hardware/software fault injection technique to corrupt bits in a computer running an

automotive-style engine control application. The conclusions of this paper start by saying: “We have demonstrated that bit-flips inside a central processing unit executing an engine control program can cause critical failures, such as permanently locking the engine’s throttle at full speed.” (Vinter 2001). Fault injection remains a preferred technique for determining whether there are data corruption vulnerabilities that can result in unsafe system behavior.

## **References:**

- Brown, D., “Solving the software safety paradox,” Embedded System Programming, December 1998, pp. 44-52.
- Kleidermacher, D. & Griglock, M., Safety-Critical Operating Systems, Embedded Systems Programming, Sept. 2001, pp. 22-36.
- Seeger, M., Fault-Tolerant Software Techniques, SAE Report 820106, International Congress & Exposition, Society of Automotive Engineers, 1982, pp. 119-125.
- Stepner, D., Nagarajan, R., & Hui, D., Embedded application design using a real-time OS, Design Automation Conference, 1999, pp. 151-156.
- Vinter, J., Aidemark, J., Folkesson, P. & Karlsson, J., Reducing critical failures for control algorithms using executable assertions and best effort recovery, International Conference on Dependable Systems and Networks, 2001, pp. 347-356.

Posted by Phil Koopman at [Monday, May 05, 2014](#) 2 comments  

Labels: [dependability](#), [error detection](#), [Essays](#)

**Monday, April 28, 2014**

## **Monitor Actuator Pair Design Pattern**

A previous post discussed patterns for safe systems, including using redundant processors that cross-check. Another accepted pattern for ensuring that there is no single point failure is a Monitor-Actuator Pair. In this architectural pattern, the "actuator" performs the control computation or other safety critical function. The "monitor" checks that the actuator is performing safely. If either the monitor or the actuator detects a problem, typically they do a mutual shut-down as with a replicated pair. The motivation to use a monitor-actuator pattern is that the monitor can often be simpler than the actuator, helping reduce system cost in both direct and indirect ways.

**Consequences:** When using a monitor-actuator safety architecture, the monitor must be able to mitigate faults without requiring that the actuator software participates in that mitigation. The consequence of implementing a monitor-actuator

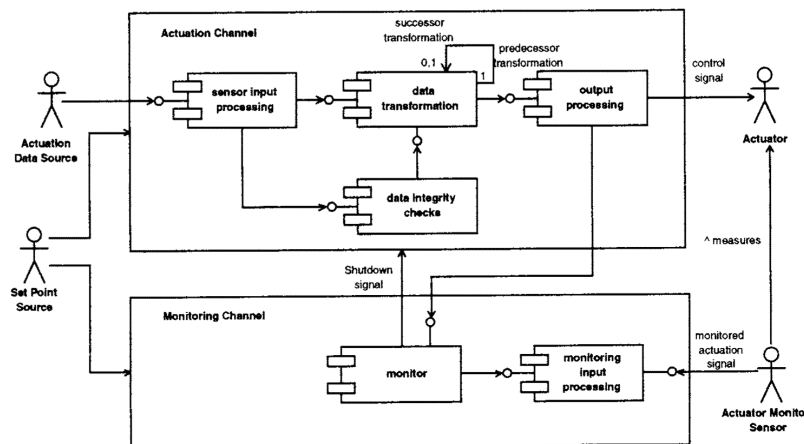
pair improperly is that when the actuator experiences a fault, that fault may not be mitigated.

### Accepted Practice:

- All monitor functions must execute on a separate microcontroller or other isolated hardware platform. This is to ensure that execution errors in the actuator cannot compromise the operation of the monitor.
- When a fault is detected, the monitor must mitigate the fault (e.g., do a system reset or close the throttle) regardless of any function performed (or not performed) by the actuator. This is to ensure that execution errors in the actuator cannot prevent fault mitigation from succeeding.

### Discussion:

A well established design technique for mitigating software errors is to have two independent hardware components operate as a “monitor-actuator” CPU pair. The actuator CPU is the component that actually performs the computation or control function. For example, the actuator might compute a throttle angle command based on accelerator position. (The name “actuator” is just a role that is played – it can include calculation and other functions.) An independent monitor Chip is used to avoid having the actuator CPU be a single point of failure. The general assumption is that the actuator CPU may fail in some detectable way, and the monitor’s job is to detect and mitigate any such failure. A common mitigation technique is resetting the actuator CPU. (Note that for the remainder of this section I use the term “actuator” for this design pattern to mean an actuator CPU, and not a physical actuation output device.)



Monitor-Actuator Design Pattern. (Douglass 2002, Section 9.6)

The monitor must be implemented as an independent microcontroller that does an acceptance test (a computation to determine if the actuator’s outputs are safe) or other computation to ensure that the actuator is operating properly. The precise check on the actuator’s output used is application specific, and multiple such checks might be appropriate for a

particular system. If the monitor detects that the actuator is not behaving in a safe manner, the monitor performs a fault mitigation function.

An example of such a monitor-actuator pair would be a throttle control microcontroller (the actuator) and an associated independent monitor microcontroller. If the throttle actuator CPU hangs or issues an unsafe throttle command, the monitor detects that condition and performs a fault mitigation action such as resetting the throttle actuator CPU. To perform this function, the monitor observes data being used by the actuator to perform computations as well as observes the outputs of the actuator. The monitor then decides if the throttle position is reasonable given the observed inputs (including, for example, brake pedal position) and resets the actuator when the checks fail. Examples of checks that would be expected in this sort of system would be a heartbeat check designed along the lines of a watchdog timer approach (ensuring the actuator is processing data periodically rather than being dead for some reason), and checking to ensure that commanded throttle position is reasonable given inputs to the actuator, such as the brake pedal position.

Proper operation of a monitor-actuator pair requires that the monitor has an ability to perform fault mitigation regardless of any execution problem that may be taking place in the actuator. For example, the monitor might use a hardware control line to reset the actuator and move the physical actuator to a safe position. Any assumption that the actuator will cooperate in fault mitigation (e.g., via a software task on the actuator accepting a reset request and initiating a reset), is considered a bad practice. Moreover, there should be no way for the actuator to inhibit the mitigation even if a software defect on the actuator actively tries to do so via faulty operation. The reason for this is that if the actuator is acting in a way that is defective, then relying on that defective component to perform any function properly (including self fault mitigation such as setting a trouble code or resetting) is a bad practice. Rather, the monitor must have a complete and independent ability to mitigate a fault in the actuator, regardless of the state of the actuator.

### **Selected Sources:**

Douglass 2002 describes this pattern under in section 9.6. Douglass summarizes the operation as: “In the Monitor-Actuator Pattern, an independent sensor maintains a watch on the actuation channel looking for an indication that the system should be commanded into its fail-safe state.” The description emphasizes the need for independence of the two components.

“For the higher integrity levels, consider using an independent monitor processor to initiate a safe state.” (MISRA Software Guidelines 3.4.1.6.h, page 36). MISRA further makes it clear that the two “channels” (the monitor and the actuator) must “provide

truly independent detection and reaction to errors” to provide safety mitigation (MISRA Report 2, p. 8, emphasis added).

Delphi’s automotive electronic throttle control system is said to use a primary processor and a redundant checking processor in keeping with this design practice, including an arrangement in which the second “processor performs redundant ETC sensor and switch reads.” (McKay 2000, pg. 8).

Safety standards also make it clear that the mere presence of a single point fault is unacceptable. For example, FAA DO-178b, the aviation software safety standard, specifically talks about a monitor/actuator pattern, saying that risk of failure is mitigated only if this condition (among several) is satisfied: “Independence of Function and Monitor: The monitor and protective mechanism are not rendered inoperative by the same failure condition that causes the hazard.” (DO 178-b Section 2.3.3.c).

### **References:**

- Douglass, B. P., Real-Time Design Patterns: robust scalable architecture for real-time systems, Pearson Education, first printing, September 2002, copyright by Pearson in 2003.
- MISRA, Development Guidelines for Vehicle Based Software, November 1994 (PDF version 1.1, January 2001).
- MISRA, Report 2: Integrity, February 1995
- McKay, D., Nichols, G. & Schreurs, B., Delphi Electronic Throttle Control Systems for Model Year 2000; Driver Features, System Security, and EOM Benefits. SAE 2000-01-0556, 2000.
- Do-178b, Software considerations in airborne systems and equipment certification, Royal Technical Commission on Aviation, Dec 1, 1992.

Posted by Phil Koopman at [Monday, April 28, 2014](#) 0 comments    
 Labels: [dependability](#), [Essays](#), [safety](#)

**Monday, April 21, 2014**

## **Layered Defenses for Safety Critical Systems**

Even if designers mitigate all single point faults in a design, there is always the possibility of some unexpected fault or combination of correlated faults that causes a system to fail in an unexpected way. Such failures should ideally never happen, but in practice no design analysis is perfectly comprehensive, especially if there are unconsidered correlations that make seemingly independent faults happen together, so they do happen. To mitigate such problems, system designers use layers of mitigation, which is a practice sometimes referred to as “defense in depth.”

### **Consequences:**

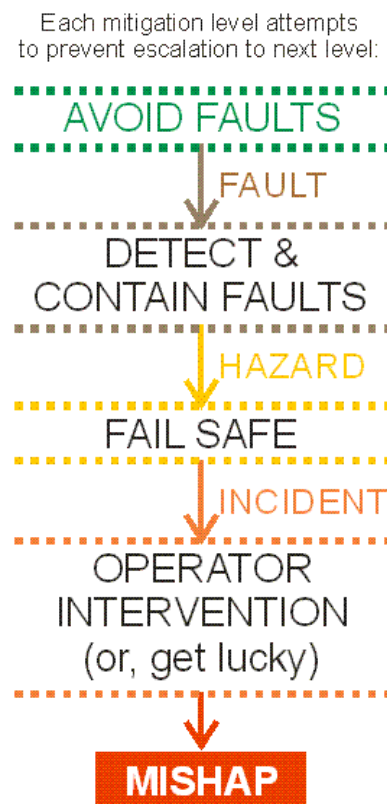
If a layered defensive strategy is defective, a failure can bypass intended mitigation strategies and result in a mishap.

## Accepted Practices:

- The accepted practice for layered systems is to ensure that no single point of failure, nor plausible combination of failures, exists which permits a mishap to occur. For layered defense purposes, a single point of failure includes even a redundant component subsystem (e.g., a 2002 redundant self-checking CPU pair might fail due to software defect present on both modules, so a layered defense provides an alternate way to recover from such a failure)
- The existence of multiple layers of protection is only effective if the net result gives complete, non-single-point-of-failure, coverage of all relevant faults.
- The goal of layered defenses should be maximizing the fraction of problems that are caught at each layer of defense to reduce the residual probability of a mishap.

## Discussion:

A layered defense system typically rests on an application of the principle of fault containment, in which a fault or its effects are contained and isolated so as to have the least effect on the system possible. The starting point for this is using fault containment regions such as [2002 systems or similar design patterns](#). But, a prudent designer admits that software faults or correlated hardware faults might occur, and therefore provides additional layers or protection.



Layered defenses attempt to prevent escalation of fault effects.

The figure above shows the general idea of layered defenses from a fault tolerant computing perspective. First, it is ideal to avoid both design and run-time faults. But, faults do crop up, and so mechanisms and architectural patterns should be in place to detect and contain those faults using fault containment regions. If a fault is not contained as intended, then the system experiences a hazard in that its primary fault tolerance approach has not worked and the system has become potentially unsafe. In other words, some fraction of faults might not be contained, and will result in hazards.

Once a hazard has manifested, a "fail-safe" mitigation strategy can help reduce the chance of a bigger problem occurring. A fail safe might, for example, be an independent safety system triggered by an electro-mechanical monitor (for example, a pressure relief valve on a water heater that releases pressure if steam forms inside the tank). In general, the system is already in an unsafe operating condition when the fail-safe has been activated. But, successful activation of a fail-safe may prevent a worse event much of the time. In other words, the hope is that most hazards will be mitigated by a fail-safe, but a few hazards may not be mitigated, and will result in incidents.

If the fail-safe is not activated then an incident occurs. An incident is a situation in which the system remains unsafe long enough for an accident to happen, but due to some combination of operator intervention or just plain luck, a loss event is avoided. In many systems it is common to catch a lucky break when the system fails, especially if well trained operators are able to find creative ways to recover the system, such as by shifting a car's transmission to neutral or turning off the ignition when a car's engine over-speeds. (It is important to note that recovering such a system doesn't mean that the system was safe; it just means that the driver had time and training to recover the situation and/or got lucky.) On the other hand, if the operator doesn't manage to recover the system, or the failure happens in a situation that is unrecoverable even by the best operator, a mishap will occur resulting in property damage, personal injury, death, or other safety loss event. (The general description of these points is based on Leveson 1986, pp. 149-150.)

A well known principle of creating safety critical systems is that hazardous behavior displayed by individual components is likely to result in an eventual accident. In other words, with a layered defense approach, components that act in a hazardous way might lead to no actual mishap most times, because a higher level safety mechanism takes over, or just because the system gets "lucky." However, the occurrence of such hazards can be expected to eventually result in an actual mishap, when some circumstance results in which the safety net mechanism fails.

For example, a fault containment might work 99.9% of the time, and fail-safes might also work 99.9% of the time. Thousands of tests might show that one or another of these safety layers saves

the day. But, eventually, assuming the probability the safety layers being effective is random and independent, both will fail for some infrequent situation, causing a mishap. (Two layers at 99.9% give unmitigated faults of:  $0.1\% * 0.1\% = 0.0001\%$ , which is unlikely to be seen in testing, but still isn't zero.) The safety concept of avoiding single point failures only works if each failure is infrequent enough that double failures are unlikely to ever happen in the entire lifetime of the operational fleet, which can be millions or even billions of hours of exposure for some systems. Doing this in practice for large deployed fleets requires identifying and correcting all situations detected in which single point failures are not immediately and completely mitigated. You need multiple layers to catch infrequent problems, but you should always design the system so that the layers are never exercised in situations that occur in practice.

### **Selected Sources:**

Most NASA space systems employ failure tolerance (as opposed to fault tolerance) to achieve an acceptable degree of safety. Failure tolerance means not only are faults tolerated within a particular component or subsystem, but the failure of an entire subsystem is tolerated. (NASA 2004 pg. 114) These are the famous NASA backup systems. "This is primarily achieved via hardware, but software is also important, because improper software design can defeat the hardware failure tolerance and vice versa." (NASA 2004 pg. 114, emphasis added)

Some of the layered defenses might be considered to be forms of graceful degradation (e.g., as described by Nace 2001 and Shelton 2002). For example, a system might revert to simple mechanical controls if a 2002 computer controller does a safety shut-down. A key challenge for graceful degradation approaches is ensuring that safety is maintained for each possible degraded configuration.

See also previous blog posting on: [Safety Requires No Single Points of Failure](#)

### **References:**

- Leveson, N., Software safety: why, what, how, Computing Surveys, Vol. 18, No. 2, June 1986, pp. 125-163.
- Nace, W. & Koopman, P., "A Graceful Degradation Framework for Distributed Embedded Systems," Workshop on Reliability in Embedded Systems (in conjunction with Symposium on Reliable Distributed Systems/SRDS-2001), October 2001.
- NASA-GB-8719.13, NASA Software Safety Guidebook, NASA Technical Standard, March 31, 2004.
- Shelton, C., & Koopman, P., "Using Architectural Properties to Model and Measure System-Wide Graceful Degradation," Workshop on Architecting Dependable Systems (affiliated with ICSE 2002), May 25 2002.

Posted by Phil Koopman at [Monday, April 21, 2014](#) 0 comments  

Labels: [Essays](#), [safety](#)

**Monday, April 14, 2014**

## **Redundant Input Processing for Safety**

Redundant analog and digital inputs to a safety critical system must be fed to independent chips to ensure no single point failure exists. (This posting is a follow-on to [a previous post about single points of failure](#).)

**Consequences:** If fully replicated input processing and validation is not implemented with complete avoidance of single points of failure, it is possible for a single fault to result in erroneous input values causing unsafe system operation.

### **Accepted Practices:**

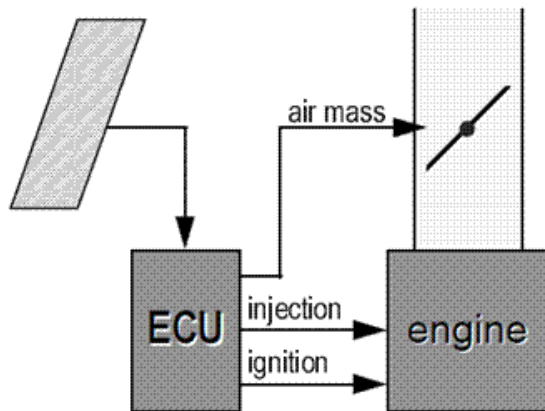
- A safety critical system must not have any single point of failure that results in a significant unsafe condition if that failure can reasonably be expected to occur during the operational life of the deployed fleet of systems. Redundant input processing is an accepted practice that can help avoid single point failures.

### **Discussion:**

A specific instance of avoiding single points of failure involves the processing of data inputs. It is imperative that safety critical input signals be duplicated and processed independently to avoid a single point of failure in input processing.

Analog inputs must be converted to digital signals via an A/D converter, which is a relatively complex apparatus that takes up a significant amount of chip area. For this reason, it is common to use a single shared A/D converter with multiple shared ("muxed") inputs to that single converter. If redundant external inputs are run through the same A/D, this creates a single point of failure in the form of the A/D converter itself and the associated control circuitry.

Mauser gives an example of this problem applied to automotive throttle control, showing that only a "true 2-channel system" (e.g., a [2002 system](#) with redundant inputs) provides safety.



- Faults in the A/D converters may cause a runaway, if both pedal sensor values are wrong. This is possible, if both analogue signal from the pedal are multiplexed to a single A/D converter; only at system #3 this fault constellation is avoided. The A/D converter errors can be recognized with a rather high probability, so that the resulting effect is rather small. But this is the only single fault that leads to the runaway case system #2b.
- System #3:** This is a kind of true 2-channel system. The processors do processor checks; in addition the 2nd processor does process checking. Both processors have A/D converters, and the redundant pedal and throttle signals are delivered to both processors.

|            | A/D converters | Comparing redundant signals | Processor check p1 | Processor check p2 | Process checking |
|------------|----------------|-----------------------------|--------------------|--------------------|------------------|
| System #1  | p1             | p1                          | none               | irrelevant         | p1               |
| System #2a | p1             | p1                          | p2                 | p1                 | none             |
| System #2b | p1             | 2x p1                       | p2                 | p1                 | p1               |
| System #3  | p1 and p2      | p1 and p2                   | p2                 | p1                 | p2               |

Table 1: Significant differences of the considered system alternatives

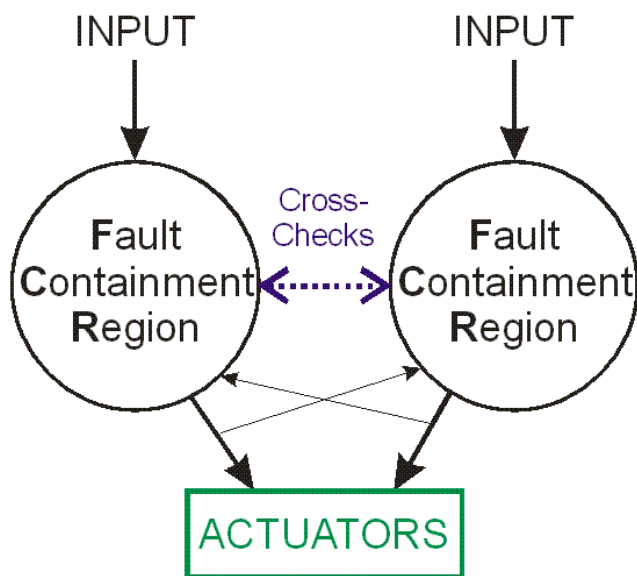
Figures from Mauser 1999, pp. 731, 738-739 showing an example throttle control system that causes runaway unless a truly redundant system (dual CPUs plus dual A/D conversion) is used.

Similarly, digital inputs that are processed in the same chip have common circuitry affecting their operation, which in typical chips includes a direction register that determines whether a digital pin is an input or output.

For both analog and digital inputs, an additional way of looking at single point failures is that if both redundant input signals are processed on the same chip, that one chip is subject to arbitrary faults, with arbitrary fault behavior including the possibility of corrupting both inputs in a way that is both faulty and undetectable by other components in the system. Unless some independent means of ensuring system safety is present, such a single point of failure impairs system safety. An arbitrary fault on a single chip that processes both copies of an analog input sensor might declare the sensor to be fully activated, but within normal

operational limits, resulting in that value being processed by the rest of the system whether the input is really active or not. For example, an embedded CPU in a car might think that the brake pedal is depressed, accelerator pedal is depressed, or parking brake is engaged despite the potential presence of redundant sensors on those controls if both sensors pass through the same A/D converter or digital input port and there is a common-mode hardware fault in those input processing circuits.

Beyond the need to independently computer results on two different chips, there is an additional requirement for safety that each of the two chips independently and fully compare the inputs to detect any faults. For example, if chips A and B both process inputs, but only chip B compares them for correctness, then there is a single point of failure if chip B has a bad input and incorrectly reports the comparison as passing (this only counts as one failure because chip B can fail in an arbitrary way in which it both mis-interprets input B and "lies" about the comparison with input A being OK). A safe way to do such a comparison is that chip A compares both inputs, chip B compares both inputs, and the system only continues operation if both chip A and chip B agree that each of their comparisons validated the inputs as being consistent. Moreover, cross-checks on the outputs based on those inputs must also be performed to detect faults that occur after input processing. That generally leads to a 2oo2 architecture like the one shown below, with each FCR usually being a CPU chip.



Sometimes both inputs go to both FCRs, but then it must be ensured that there is hardware isolation in place so that a hardware fault on one FCR can't propagate to the inputs of the other FCR via the shared input lines. Another complication is that redundant sensors often do not produce identical output values, and the problem of determining distributed agreement turns out to be very difficult even if all you need is an approximate agreement result. In general, once you have replication, getting agreement across the replicated copies of

inputs and computations requires some effort (Poledna 1995). But, these are the sorts of issues that engineers routinely work through when creating safety-critical systems.

In the end, having a single shared A/D converter or other input circuit for a safety critical system is inadequate. You must have two separate input processing circuits on two separate chips to have two independent Fault Containment Regions (e.g., using a 2002 architectural approach with redundant inputs). This is required to achieve safety for a high-integrity application, and any high-integrity embedded system that uses a shared A/D converter on a single chip to process redundant inputs is unsafe.

### **References:**

- Mauser, Electronic throttle control – a dependability case study, J. Univ. Computer Science, 5(10), 1999, pp. 730-741.
- Poledna, S., "Fault tolerance in safety critical automotive applications: cost of agreement as a limiting factor ", Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on , 27-30 Jun 1995 Page(s): 73 -8

Posted by Phil Koopman at [Monday, April 14, 2014](#) 0 comments  

Labels: [architecture](#), [Essays](#), [safety](#)

**Monday, April 7, 2014**

## **Self-Monitoring and Single Points of Failure**

[A previous post discussed single points of failure in general.](#)

Creating a safety-critical embedded system requires avoiding single points of failure in both hardware in software. This post is the first part of a discussion about examples of single points of failure in safety critical embedded systems.

**Consequences:** A consequence of having a single point of failure is that when a critical single point fails, the system becomes unsafe via taking an unsafe action or ceasing to perform critical functions.

**Accepted Practices:** The following are accepted practices for avoiding single point failures in safety critical systems:

- A safety critical system must not have any single point of failure that results in a significant unsafe condition if that failure can reasonably be expected to occur during the operational life of the deployed fleet of systems. Because of their high production volume and usage hours, for automobiles, aircraft, and similar systems it must be expected that any single microcontroller hardware chip and

software on any single chip will fail in an arbitrarily unsafe manner.

- Properly implemented monitor-actuator pairs, redundant input processing, and a comprehensive defense-in-depth strategy are all accepted practices for mitigating single point faults (see future blog entries for postings on those topics).
- Multiple points of failure that can fail at the same time due to the same cause, can accumulate without being detected and mitigated during system operation, or are otherwise likely to fail concurrently, must be treated as having the same severity as a single point of failure.

### **Discussion:**

MISRA Report 2 states that the objective of risk assessment is to “show that no single point of failure within the system can lead to a potentially unsafe state, in particular for the higher Integrity Levels.” (MISRA Report 2, 1995, pg. 17). In this context, “higher Integrity levels” are those functions that could cause significant unsafe behavior, typically involving passenger deaths. That report also says that the risk from multiple faults must be sufficiently low to be acceptable.

Mauser reports on a Siemens Automotive study of electronic throttle control for automobiles (Mauser 1999). The study specifically accounted for random faults (id. p. 732), as well as considering the probability of a “runaway” incidents (id., p. 734) in which an open throttle fault could cause a mishap. It found a possibility of single point failures, and in particular identified dual redundant throttle electrical signals being read by a single shared (multiplexed) analog to digital converter in the CPU (id., p. 739) as a critical flaw.

Ademaj says that “independent fault containment regions must be implemented in separate silicon dies.” (Ademaj 2003, p. 5) In other words, any two functions on the same silicon die are subject to arbitrary faults and constitute a single point of failure.

But Ademaj didn’t just say it – he proved it via experimentation on a communication chip specifically designed for safety critical automotive drive-by-wire applications (id., pg. 9 conclusions), and those results required the designers of the TTP protocol chip (based on the work of Prof. Kopetz) to change their approach to achieving fault tolerance to the use of a Star topology because combining a network CPU with the network monitor on the same silicon die was proven to be susceptible to single points of failure even though the die had been specifically designed to physically isolate their network monitor from their main CPU. Even though every attempt had been made for on-chip isolation, two completely independent circuits sharing the same chip were observed to fail together from a single fault in a safety-critical automotive drive-by-wire design.

A fallacy in designing safety critical systems is thinking that

partial redundancy in the form of "fail-safe" hardware or software will catch all problems without taking into account the need for complete isolation of the potentially faulty component and the mitigation component. If both the mitigation and the fault are in the same Fault Containment Region (FCR), then the system can't be made entirely safe.

To give a more concrete example, consider a single CPU with a self-monitoring feature that has hardware and/or software that detects faults within that same CPU. One could envision such a system signaling to an outside device a self-health report. Such a design pattern is sometimes called a "simplex system with disengagement monitor" and uses "Built-In Test" (BIT) to do the self-checking. (Note that BIT is a generic term for self-checks, and does not necessarily mean a manufacturing gate-level test or other specific diagnostic.) If self-health checks are false, then the system fails over to a safe state via, for example, shutting down (if shutting down is safe). To be sure, doing this is better than doing nothing. But, it can never get complete coverage. What if the self-health check is compromised by the fault in the chip?

A look at a research paper on aerospace fault tolerant architectures explains why a simplex (single-FCR) system with BIT is inadequate for high-integrity safety-critical systems. Hammett (2001) figure 5 shows a simplex computer with BIT disengagement features, and says that they "increase the likelihood the computer will fail passive rather than fail active. But it is important to realize that ***it is impossible to design BIT that can detect all types of computer failures*** and very difficult to accurately estimate BIT effectiveness." (id., pg. 1.C.5-4, emphasis added) Such an architecture is said to "Fail Active" after some failures (id., Table 1, p. 1.C.5-7), where "A fail active condition is when the outputs to actuators are active, but uncontrolled. ... A fail active condition is a *system malfunction* rather than a *loss of function*." (id., pg. 1.C.5-2, emphasis per original) "For some systems, an annunciated *loss of function* is an acceptable fail-safe, but a *malfunction* could be catastrophic." (id., p. 1.C.5-3, emphasis per original) In particular, with such an architecture depending on the fraction of failures caught (which is not 100%), some "*failures will be undetected and the system may fail to a potentially hazardous fail active condition*." (id., p. 1.C.5-4, emphasis added).

Although a simplex control cannot prevent a *loss of function* failure, features can be added to minimize the probability of a *malfunction*. Figure 5 shows a simplex control that includes self-test software to allow it to detect its own failures, a Watch Dog Timer (WDT) to detect certain failures where the computer stops cycling correctly, and a

hardware output enable that will positively disengage a malfunctioning computer from the system. These Built-In-Test (BIT) features increase the likelihood that the computer will fail passive rather than fail active.

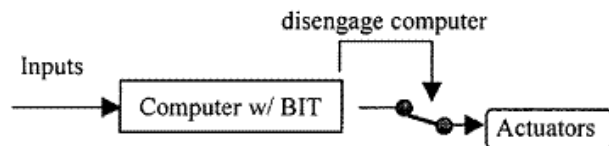


**Figure 5. Simplex, Disengagement Features**

For some systems, an annunciated *loss of function* is an acceptable fail-safe condition, but a *malfunction* could be catastrophic.

But it is important to design B computer failures and estimate BIT effectiveness. defines fault coverage detected (by BIT) to the expressed as a percentage. actual fault coverage is difficult and inexact, typically designed systems, supported by experience, are about 50%. type of system, BIT will system has failed approximately 20 failures and disengaged fail passive condition. undetected and the system hazardous fail active condition.

Although the objective system is to allow the system (FO), the computers used fail in one of two ways: Active (FA).

A fail active condition is also be described as a hazardous fail active condition is also are active, but uncontrolled.

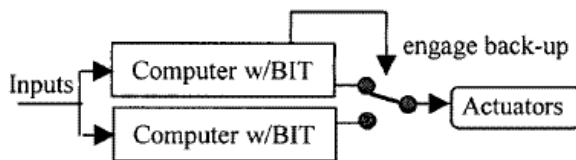
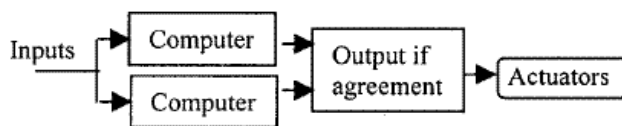
A fail active *malfunction* rather than

Table 1 from Hammett 2001, below, shows where Simplex with BIT stands in terms of fault tolerance capability. It will fail active (i.e., fail dangerously) for some single point failures, and that's a problem for safety critical systems. .

**Table 1. Fault-Tolerant Characteristics of Different System A**

| Fault-Tolerant Architecture Type                                                                                                        | Fails Operational (FO)                                                      | Fails Passive (FP)                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Simplex                                                                                                                                 | Never                                                                       | System may FP or FA a                                                                   |
| Simplex with BIT                                                                                                                        | Never                                                                       | After <u>most</u> 1 <sup>st</sup> failures (see note 1)                                 |
| Dual standby                                                                                                                            | After <u>most</u> 1 <sup>st</sup> failures                                  | After <u>most</u> 2 <sup>nd</sup> failures                                              |
| Self-checking pair                                                                                                                      | Never                                                                       | After all 1 <sup>st</sup> failures                                                      |
| Self-checking pair, simplex fault down                                                                                                  | After <u>most</u> 1 <sup>st</sup> failures                                  | After <u>all</u> 1 <sup>st</sup> failures<br>After <u>most</u> 2 <sup>nd</sup> failures |
| Dual self-checking pair                                                                                                                 | Always after 1 <sup>st</sup> failure                                        | After all 2 <sup>nd</sup> failures                                                      |
| TMR, fault down to self-checking pair                                                                                                   | Always after 1 <sup>st</sup> failure                                        | After all 2 <sup>nd</sup> failures                                                      |
| TMR, fault down to simplex                                                                                                              | Always after 1 <sup>st</sup> failure, usually after 2 <sup>nd</sup> failure | After all 2 <sup>nd</sup> failures,<br>After most 3 <sup>rd</sup> failures              |
| Note 1: <u>Most</u> means those failures that are detected by BIT. For 95% effective will fail passive following 19 out of 20 failures. |                                                                             |                                                                                         |
| Note 2: <u>Some</u> failures means those <u>not</u> detected by BIT. For 95% effective BIT, fail active following 1 in 20 failures.     |                                                                             |                                                                                         |

Note that dual standby redundancy is also inadequate even though it has two copies of the same computer with the same software. This is because the primary has to self-diagnose that it has a problem before it switches to the backup computer (Hammett Fig. 6, below). If the primary doesn't properly self-diagnose, it never switches over, resulting in a fail-active (dangerous system).

**Figure 6. Dual Standby FT Architecture****Figure 7. Self-Checking Pair**

On the other hand, a self-checking pair (Hammett figure 7 above), sometimes known as a "2 out of 2" or 2002 system, can tolerate all single point faults the following way. Each of the computers in a 2002 pair runs the same software on identical hardware, usually operating in lockstep. If the outputs don't agree, then the system disables its outputs. Any single failure that

affects the computation will, by definition, cause the outputs to disagree (because it can only affect one of the 2 computers, and if it doesn't change the output then it is not affecting the result of the computation). Most dual-point failures will also be detected, except for dual point failures that happen to affect both computers in exactly the same way. Because the two computers are separate FCRs, this is unlikely unless there is a correlated fault such as a software defect or hardware design defect. In practice, the inputs are also replicated to avoid a bad sensor being a single point of failure as well (Hammett's figure is non-specific about inputs, because the focus is on computing patterns). 2002 is not a free lunch in many regards, and I'll queue a discussion of the gory details for a future blog post if there is interest. Suffice it to say that you have to pay attention to many details to get this right. But it is definitely possible to build such a system.

With a 2002 system, the second CPU does not improve availability, but in fact reduces it because there are twice as many computers to fail. To attain availability, a redundant failover set of 2002 computers can be used (Hammett Fig.9 -- dual self-checking pair). And in fact this is a commonly used architecture in railway switching equipment. Each 2002 pair self-checks, and if it detects an error it shuts down, swapping in the other 2002 pair. So having a single 2002 pair is done for safety. The second 2002 pair is there to prevent outages (see Hammett figure 9, below).

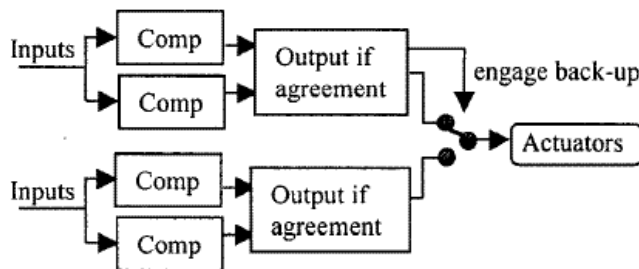


**Figure 9. Dual Self-Checking Pair**

From the above we can see that avoiding single points of failure requires at least two CPUs, with care taken to ensure that each CPU is a separate fault containment region. If you need a fail-operational system, then 4 CPUs arranged per figure 9 above will give you that, but at a cost of 4 CPUs.

Note that we have not at any point attempted to identify some "realistic" way in which a computer can both produce a dangerous output and cause its BIT to fail. Such analysis is not required when building a safe system. Rather, the effects of failure modes in electronics are more subtle and complex than can be readily understood (and some would argue that many real but infrequent failure modes are too complex for anyone to understand). It is folly to try to guess all possible failures and somehow ensure that the BIT will never fail. But even if we tried

to do this, the price for getting it wrong in terms of death and destruction with a safety critical system is simply too high to take that chance. Instead, we simply assert that Murphy will find a way to make a simplex system with BIT fail active, and take that as a given.

By way of analogy, there is no point doing analysis down to single lines of code or bolt tensile strengths in high-vibration environments within a jet engine to know that flying across the Pacific Ocean in a jet airliner with only one engine working at takeoff is a bad idea. Even perfectly designed jet engines break, and any single copy of perfectly design jet engine software will eventually fail (due to a single event upset within the CPU it is running in, if for no other reason). The only way to achieve safety is to have true redundancy, with no single point failure whatsoever that can possibly keep the system from entering a safe state.

In practice the "output if agreement" block shown in these figures can itself be a single point of failure. This is resolved in practical systems by, for example, having each of the computers in a 2002 pair control the reset/shutdown line on the other computer in the 2002 pair. If either computer detects a mismatch, it both shuts down the other CPU and commits suicide itself, taking down the pair. This system reset also causes the switch in a dual 2002 system to change over to the backup pair of computers. And yes, that switch can also be a single point of failure, which can be resolved by for example having redundant actuators that are de-energized when the owner 2002 pair shuts down. And, we have to make sure our software doesn't cause correlated faults between pairs by ensuring it is of sufficiently high integrity as well.

As you can see, flushing out single points of failure is no small thing. But if you want to build a safety critical system, getting rid of single points of failure is the price of admission to the game. And that price includes truly redundant CPUs for performing safety critical computations.

### **References:**

- Hammett, Design by extrapolation: an evaluation of fault-tolerant avionics, 20th Conference on Digital Avionics Systems, IEEE, 2001.
- MISRA, Report 2: Integrity, February 1995.
- Mauser, Electronic throttle control – a dependability case study, J. Univ. Computer Science, 5(10), 1999, pp. 730-741.

Posted by Phil Koopman at [Monday, April 07, 2014](#) 0 comments  

Labels: [dependability](#), [Essays](#), [hardware](#), [safety](#), [system design](#)

**Monday, March 31, 2014**

## **How Often Will Random Faults Kill Someone?**

If you design a system that doesn't eliminate all single points of failure, eventually your safety-critical system will kill someone. How often is that? Depends on your exposure, but in real-world systems it could easily be on a daily basis.

### **"Realistic" Events Happen Every Million Hours**

Understanding software safety requires reasoning about extremely low probability events, which can often lead to results that are not entirely intuitive. Let's attempt to put into perspective where to set the bar for what is "realistic" and why single point fault vulnerabilities are inherently dangerous for systems that can kill people if they fail.

If a human being lives to be 90 years old, that is 90 years \* 365.25 days/yr \* 24 hrs/day = 788,940 hours. While the notion of what a "realistic" fault is can be subjective, perhaps an individual will think something is a realistic failure type if (s)he can expect to see it happen in one human lifetime.

This definition has some problems since everyone's experience varies. For example, some would say that total loss of dual redundant hydraulic service brakes in a car is unrealistic. But that has actually happened to me due to a common-mode cracking mechanical failure in the brake fluid reservoir of my vehicle, and I had to stop my vehicle using my parking brake. So I'd have to call loss of both service brake hydraulic systems a realistic fault from my point of view. But I have met automotive engineers who have told me it is "impossible" for this failure to happen. The bottom line: intuition as to what is "realistic" isn't enough for these types of matters -- you have to crunch the numbers and take into account that if you haven't seen it yourself that doesn't mean it can't happen.

So perhaps it is also realistic if a friend has told you a story about such a fault. From a probability point of view let's just say it's "realistic" if it is likely to happen every 1,000,000 hours (directly to you in your life, plus 25% extra to account for second-hand stories).

Probability math in the safety critical system area is usually only concerned with rounded powers of ten, and 1 million is just a rounding up of a human lifespan. Put another way, if it happens once per million hours, let's say it's "realistic." By way of contrast, the odds of winning the top jackpot in Powerball are about one in 175 million ([http://www.powerball.com/powerball/pb\\_prizes.asp](http://www.powerball.com/powerball/pb_prizes.asp)), and someone eventually wins without buying 175 tickets per hour for an entire million-hour human lifetime, so a case can be made for much less frequent events to also be "realistic" -- at least for someone in the population.

I should note at this point that aircraft safety and train safety are at least 1000 times more rigorous than one catastrophic failure per million hours. Then again, trains and jumbo jets hold hundreds of people, who are all being exposed to risk together.

So the point of this essay is simply to make safety probability more human-accessible, and not to argue that one catastrophic fault per million hours is OK for a high criticality system -- it's not!

### **"Realistic" Catastrophic Failures Will Happen To A Deployed Fleet Daily**

Obermaisser gives permanent hardware failure rates as about 100 FIT (Obermaisser p. 10 in the context of drive-by-wire automobiles. Note that 1 "FIT" is 1 failure per billion hours, so 100 FIT is one failure per 10 million operating hours). This means that you aren't likely to see any *particular* car component failure in your lifetime. But cars have lots of components all failing at this rate, so there is a reasonable chance you'll see *some* component failure and need to buy a replacement part. (These failure rates are for normal operating lifetimes, and do not count the fact that failures are more frequent as parts near end of life due to age and wearout.)

However, transient failures, such as those caused by cosmic rays, voltage surges, and so on, are much more common, reaching rates of 10,000-100,000 FIT (Obermaisser p. 10; 100,000 per billion hours = 100 per million hours). That means that any particular component will fail in a way that can be fixed by rebooting or the like about 10 to 100 times per million hours -- well within the realm of realistic by individual human standards. Obermaisser also gives a frequency for arbitrary faults as  $1/50^{\text{th}}$  of all faults (id., p. 8). Thus, any particular component, can be expected to fail 10 to 100 times per million hours, and to do so in an "arbitrary" way (which is software safety lingo for "dangerous") about 0.2 to 2 times per million hours. This is right around the range of "realistic," straddling the once per million hours frequency.

In other words, if you own one single electronic device your whole life, it will suffer a potentially dangerous failure about once in your lifetime. If you continuously own vehicles for your entire life, and they each have 100 computer chips in them, then you can expect to see about one potentially dangerous failure per year since there are 100 of them to fail -- and whether any such failure actually kills you depends upon how well the car's fault tolerance architecture successfully deals with that failure and how lucky you are that day. (We'll get into the point that you don't drive your car 24x7 momentarily.) That is why it is so important to have redundancy in safety-critical automotive systems. These are general examples to indicate the scale of the issues at hand, and not specifically intended to correspond to exact numbers in any particular vehicle, especially one in which failsafes are likely to

somewhat reduce the mishap rate by providing partial redundancy.

Now let's see how the probabilities work when you take into account the large size of the fleet of deployed vehicles. Let's say a car company sells 1 million vehicles in a year. Let's also say that an average vehicle is driven about 1 hour per day (FHA 2009 National Household Travel Survey, p. 31 says 56 minutes per day, but we're rounding off here). That's 1 million hours per day. If we multiply that by the range of dangerous transient faults per component (0.2 to 2 per million hours), that means that the fleet of vehicles can expect 0.2 to 2 dangerous transient faults per day for any particular safety critical control component in that fleet. And there are likely more than one safety critical components in each car. In other words, while a dangerous failure may seem unlikely in an individual basis, designers must expect dangerous failures to happen on a daily basis in a large deployed fleet.

While the exact numbers in this calculation are estimates, the important point is that a competent safety-critical designer must take into account arbitrary failures when designing a system for a large deployed fleet.

The above is an example calculation as to why *redundancy* is required to achieve safety. Arbitrary faults can be expected to be dangerous (we've already thrown away 98% of faults as benign in these calculations – we're just keeping 2% of faults as dangerous faults). Multiple fault containment regions (FCRs) must be used to mitigate such faults.

It is important to note that in the fault tolerant and safety critical computing fields “arbitrary” means just that – it is a completely unconstrained failure mode. It is not simply a failure that seems “realistic” based on some set of preconceived notions or experiences. Rather, designers consider it to be the worst possible failure of a single chip in which it does the worst possible thing to make the system unsafe. For example, a pyrotechnic device must be expected to fire accidentally at some point unless there is true redundancy that mitigates every possible single point of failure, whether the exact failure mechanism in the chip can be imagined by an engineer or not. The only way to avoid single point failures is via some form of true redundancy that serves as an independent check and balance on safety critical functions.

As a second source for similar failure rate numbers, Kopetz, who specializes in automotive drive-by-wire fault tolerant computer systems, gives an acceptable mean-time-to-failure of safety-critical applications of one critical failure per 1 billion hours (more than 100,000 years for any particular vehicle), saying that the “dependability requirements for a drive-by-wire system are even more stringent than the dependability requirements for a fly-by-wire systems, since the number of exposed hours of humans is higher in the automotive domain.” (Kopetz 2004, p.

32). This cannot be achieved using any simplex hardware scheme, and thus requires redundancy to achieve that safety goal. He gives expected component failure rates as 100 FIT for permanent faults and 100,000 FIT for transient faults (id., p. 37).

By the same token, any first fault that persists a long time during operation without being detected or mitigated sets the stage for a second fault to happen, resulting in a sequence of faults that is likewise unsafe. For example, consider if a vehicle has a manufacturing fault or other fault that is undetected but disables a redundant sensor input without that failure being detected or fixed. From the time that fault happens forward, the vehicle is placed at the same risk as if it only had a single point failure vulnerability in the first place. Redundancy doesn't do any good if a redundant component fails, nobody knows about it, nobody fixes it, and the vehicle keeps operating as if nothing is wrong. For example, for this reason a passenger jet with two engines isn't allowed to take off on an over-ocean flight with only one engine working at takeoff.

### **Software Faults Only Make Things Worse**

The above fault calculations "assume the absence of software faults," (Obermaisser, p. 13), which is an unsupported assumption for many systems. Software faults can only be expected to make the arbitrary failure rate worse. Having software-implemented failsafes and partial redundancy may mitigate dangerous faults to more than the computed 2%. But it is impossible for a mitigation technique in the same fault containment region as a failure to be successful at mitigating all possible failure modes.

Software defects manifest as single points of failure in a way that may be counter-intuitive. Software defects are design defects rather than run-time hardware failures, and are therefore present all the time in every copy of a system that is created. (MISRA Report 2, p. 7) Moreover, in the absence of perfect memory and temporal isolation mechanisms, every line of software in a system has the potential to affect the execution of every other line in the system. This is especially problematic in systems which do not use memory protection, which do not have an adequate real time scheduling approach, which make extensive use of global variables, and have other similar problems that result in an elevated risk of the activation of one software defect causing system-wide problems, or causing a cascading activation of other software defects. Therefore, in the absence of compelling proof of complete isolation of software tasks from each other, the entire CPU must be considered a single FCR for software, making the entirety of software on a CPU a single point of failure for which any possible unsafe behavior must be fully and completely mitigated by mechanisms other than software on that same CPU.

Some examples of ways in which multiple apparent software defects might result within a single FCR but still constitute a

single point of failure include: corruption of a block of memory (corrupting the values of many global variables), corruption of task scheduling information in and operating system (potentially killing or altering the execution patterns of several tasks), and timing problems that cause multiple tasks to miss their deadlines. It should be understood that these are merely examples – arbitrarily bad faults are possible from even a seemingly trivial software defect.

MISRA states that FMEA and Fault Tree Analysis that examine specific sources and effects of faults are “not applicable to programmable and complex non-programmable systems because it is impossible to evaluate the very high number of possible failure modes and their resulting effects.” (MISRA Report 2 p. 17). Instead, for any complex electronics (whether or not it contains software), MISRA tells designers to “consider faults at the module level.” In other words, MISRA describes considering the entire integrated circuit for a microcontroller as a single FCR unless there is a very strong isolation argument (and considering the findings of Ademaaj (2003), it is difficult to see how that can be done).

### **Deployed Systems Will See Dangerous Random Faults**

The point of all this is to demonstrate that arbitrary dangerous single point faults can be expected to happen on a regular basis when hundreds of thousands of embedded systems such as cars are involved. True redundancy is required to avoid weekly mishaps on any full-scale production vehicle that involves hundreds of thousands of units in the field. No single point fault, no matter how obscure or seemingly unlikely, can be tolerated in a safety critical system. Moreover, multiple point faults cannot be tolerated if they are sufficiently likely to happen via accumulation of undetected faults over time. In particular, software-implemented countermeasures that run on a CPU aren't going to be 100% effective if that same CPU is the one that suffered a fault in the first place. True redundancy is required to achieve safety from catastrophic failures for large deployed fleets in the face of random faults.

It is NOT acceptable practice to start arguing whether any particular fault is “realistic” given a particular design of within a single point fault region such as an integrated circuit. This is in part because designer intuition is fallible for very low (but still important) probability events, and also in part because it is as a practical matter impossible to envision all arbitrary faults that might cause a safety problem. Rather, accepted practice is to simply assert that the worst case possible single-point failure will find a way to happen, and design to ensure that such an event does not render the system unsafe.

### **References:**

- Ademaj et al., Evaluation of fault handling of the time-triggered architecture with bus and star topology, DSN 2003.
- Kopetz, H., On the fault hypothesis for a safety-critical real-time system, ASWSD 2004, LNCS 4147, pp. 31-42, 2006.
- MISRA, Report 2: Integrity, February 1995.
- Obermaisser, A Fault Hypothesis for Integrated Architectures, International Workshop on Intelligent Solutions in Embedded Systems, June 2006, pp. 1-18.

Posted by Phil Koopman at [Monday, March 31, 2014](#) 0 comments    
Labels: [dependability](#), [Essays](#), [safety](#)

[Newer Posts](#)[Home](#)[Older Posts](#)

Subscribe to: [Posts \(Atom\)](#)

## Follow Me On Twitter

Follow @BetterEmbSw 

## Featured Post

### Top 5 Embedded Software Problem Areas

The biggest problems I see in industry code reviews are code complexity, real time performance, code quality, weak development process, and...



## Search This Blog

## Top Posts

- [Embedded Training Videos](#)
- [Top Five Embedded Problems](#)
- [Toyota Unintended Acceleration](#)
- [Best CRC Polynomials](#)

- Embedded Software Cost
- Cyclomatic Complexity
- Watchdog Timer Use
- EEPROM Corruption ✂

## Subscribe!

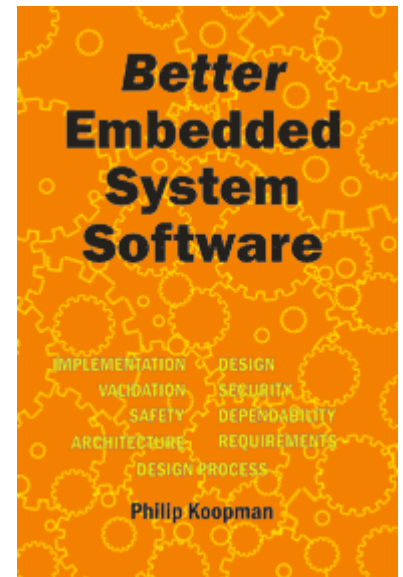
 Posts 

 All Comments ✂

## Book Info

[Click here to buy the book](#)

Free Shipping to US  
International sales via  
Amazon.com



[Click here for book description.](#)

This book gives concrete approaches to achieving better embedded software in 29 different areas.

[Click here for Free Sample Chapter](#) ✂

## Post Topics

agile methods (1)  
architecture (4)  
book (9)  
CAN (3)  
coding style (24)

[compilers](#) (2)  
[concurrency](#) (2)  
[CRC/checksum](#) (5)  
[defects](#) (9)  
[dependability](#) (21)  
[documentation](#) (1)  
[economics](#) (8)  
[Edge Case Research](#) (6)  
[editorial](#) (2)  
[EEPROM](#) (2)  
[embedded C](#) (6)  
[embedded networks](#) (4)  
[error detection](#) (10)  
[Essays](#) (23)  
[exceptions](#) (5)  
[Free Sample Chapter](#) (1)  
[globals](#) (6)  
[hardware](#) (4)  
[Internet of Things \(IoT\)](#) (1)  
[interrupts](#) (4)  
[management](#) (7)  
[math](#) (2)  
[metrics](#) (7)  
[ML validation](#) (1)  
[open source](#) (1)  
[optimization](#) (6)  
[outsourcing](#) (1)  
[peer reviews](#) (15)  
[real time](#) (13)  
[requirements](#) (4)  
[research](#) (2)  
[risk management](#) (15)  
[robustness](#) (1)  
[safety](#) (41)  
[security](#) (14)  
[self-driving cars](#) (10)  
[SQA](#) (3)  
[system design](#) (5)  
[testing](#) (14)  
[time keeping](#) (3)  
[tools](#) (2)  
[Toyota UA](#) (2)  
[traceability](#) (1)  
[travel](#) (1)  
[tutorial](#) (12)  
[updates](#) (1)  
[USI](#) (3)  
[versioning](#) (3)  
[watchdog timer](#) (9)



## About Me

Phil Koopman



I've done embedded systems for big industry, the US military, startup companies, and now Carnegie Mellon University. I'm the author of the book [Better Embedded System Software](#), which goes into more detail on most of the topics discussed in my corresponding blog.

As with any blog, these posts often contain speculative and partially formed thoughts, and should not be interpreted as a fully considered opinion unless stated otherwise.

Key pages:

[Academic home page at CMU](#)

[Embedded Software Blog](#)

[Checksum and CRC Blog](#)

[View my complete profile](#)

## Blog Archive

- ▼ [2018](#) (2)
  - ▼ [February](#) (1)
    - [Robustness Testing of Autonomy Software \(ASTAA Pap...](#)
  - [January](#) (1)
- [2017](#) (23)
- [2016](#) (26)
- [2015](#) (10)
- [2014](#) (28)
- [2013](#) (12)
- [2012](#) (10)
- [2011](#) (18)
- [2010](#) (41)



## Other Embedded Blogs

[Embedded FM Podcast](#)  
[Checksum and CRC](#)  
[Central](#)

[Michael Barr](#)  
[Nigel Jones](#)  
[Jack Ganssle](#)  
[John Regehr's Blog](#)  
[Embedded.com Columns](#)  
[Embedded Related](#)



**Buy Via Amazon**



**Reader Locations**



**Feedjit**

[Feedjit Live Blog Stats](#)